UNCLASSIFIED

Copy 14 of 35 copies

AD-A196 633

(2)

DTIC FILE COPY

IDA MEMORANDUM REPORT M-366

DEFENSE LOGISTICS AGENCY DATA SYSTEM CENTER FORMS MANAGEMENT SYSTEM (FMS)

James Wolfe
David Carney
Audrey Hook

September 1987

Prepared for
Office of the Under Secretary of Defense for Research and Engineering

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

UNCLASSIFIED

IDA Log No. HG 87-32728

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE AD-A196 633

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY 2b DECLASSIFICATION/DOWNGRADING SCHEDULE	3 DISTRIBUTION/AVAILABILITY OF REPORT Public release/distribution unlimited.
4 PERFORMING ORGANIZATION REPORT NUMBER(S) IDA Memorandum Report M-366	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION OUSDA, DIMO
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311
8a NAME OF FUNDING/SPONSORING ORGANIZATION Defense Logistics Agency 8b OFFICE SYMBOL (if applicable) DLA-ZWS	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) Cameron Station Alexandria, VA 22304-6100	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-T5-423 / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
Defense Logistics Agency Data System Center Forms Management System (FMS) (U)

12 PERSONAL AUTHOR(S)
James Wolfe, David J. Carney, Audrey A. Hook

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM ______ TO ______	1987 September	54

16 SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Ada programming language; prototyping; Forms Management System (FMS); software design; applications programming; database management system (DBMS); automated data processing (ADP); screen design.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

IDA Memorandum Report M-366 is a design specification for a general purpose forms management system (FMS) to be used at the Defense Logistics Agency (DLA) Systems Automation Center (DSAC). The design specification has been written in Ada to demonstrate the use of Ada in applications design and to provide a modular language that can be partially implemented if desired. The functional requirements reflect IDA's understanding of how terminal displays are used in several of the DLA systems. The Forms Management System (FMS) is required to operate on VT-100 and IBM 3270 terminals and IBM PC-compatible workstations.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Audrey A. Hook	22b TELEPHONE (Include area code) (703) 824-5501	22c OFFICE SYMBOL IDA/CSED

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

IDA MEMORANDUM REPORT M-366

DEFENSE LOGISTICS AGENCY DATA SYSTEM CENTER FORMS MANAGEMENT SYSTEM (FMS)

James Wolfe
David Carney
Audrey Hook

September 1987

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-T5-423

UNCLASSIFIED

TABLE OF CONTENTS

UNCLASSIFIED

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

UNCLASSIFIED

UNCLASSIFIED

LIST OF TABLES

PREFACE

The purpose of IDA Memorandum Report M-366 Defense Logistics Agency Data System Center Forms Management System, is to record substantive technical work on the functional requirements and design specifications for a prototype Ada[1] application. This work partially fulfills requirements of IDA Task T-T5-423, Defense Logistics Information System.

Mr. Bill Brykczynski, Dr. Joseph Linn, Ms. Katydean Price, and Dr. Robert Winner have reviewed this paper.

[1] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

1.0 FUNCTIONAL REQUIREMENTS FOR FORMS MANAGEMENT SYSTEM

1.1 *PURPOSE*

This IDA Memorandum Report documents the functional requirements and design specifications for a general purpose forms management system. The functional requirements reflect IDA's understanding of how terminal displays are used in several of the Defense Logistics Information systems. The design specification has been written in the Ada programming language to demonstrate the use of Ada in applications design and to provide a modular design that can be partially implemented if desired. In addition, the design specification is intended to produce a capability that can be adapted to many applications.

1.2 BACKGROUND

Computer programmers at the Defense Logistics Agency (DLA) Systems Automation Center (DSAC) must write programs to display and process the data entered on forms. These programs are application specific, terminal and computer system specific and are not generally adaptable to new hardware and software applications. The vendor software available to assist application programmers in designing screens is tightly coupled with the vendor's other products (e.g., teleprocessing monitor or a database management system (DBMS)), is supplied as object code so that it cannot be modified or tailored by the customer, and is not usable in environments that the vendor does not support with bundled products. It is difficult or impossible to recover an investment in designed forms when competitive procurement results in an award to another vendor.

As part of the Ada demonstration project, there is a need to demonstrate that an Ada tool can be used by programmers to create, modify, and maintain the forms that are used for transactions related to the logistics business.

1.3 REQUIREMENTS

The majority of Logistics Information Systems are transaction driven. In the context of these information systems, a transaction can be one of the following:

- one or more pre-defined forms that are completed by a functional person that results in another functional person taking some action, such as filling an order and/or completing another form or set of forms;
- a pre-defined form that is completed by a functional person who needs information from a data base or from another functional person; or,
- an electronic message in a standard format (e.g., MILSTRIP) that is received by a functional person who must respond to it with a set of actions that includes using other pre-defined forms.

Requirements for a Forms Management System have been grouped into high level requirements and lower level requirements, both placing constraints on the design of this system.

The high level requirements are determined by two types of users, an automated data processing (ADP) user who designs screens and implements application programs using them, and the functional user who uses the screens for transactions such as data entry, data retrieval, and messages.

1.3.1 High Level Requirements

The Forms Management System must fulfill the following high level requirements:

- Be a set of Ada packages which are compilable into an Ada program library.
- Be usable for Ada demonstration projects and adaptable to a prototype Ada application.
- Provide functions to create, delete, and modify menu-type screens.
- Provide functions to invoke menu-type screens from applications.
- Provide functions to create, delete, and modify form-type screens.
- Provide functions to gather data input from form-type screens and to pass this data, through a defined interface, to application programs written in Ada or in COBOL.

- Perform a limited but critical amount of input data verification. This verification must be performed with the same rigor for all terminal types (that is, application programs must be assured of the same verification, or lack of it, without knowing the terminal type in use).
- Be time efficient such that the functional user sees no degradation of performance from the currently available programs.
- Be designed so that all or portions of it can be developed by the programmers at Defense Logistics Agency Systems Automation Center (DSAC) to demonstrate their skills in software engineering and the Ada language.
- Be compatible with the technical environment (e.g., terminals) already in use at DSAC.

1.3.2 Lower Level Requirements

The Forms Management System supports a database of forms that must be:

- Indexed by application and by type.
- Linked with data validation programs (when they exist).
- Linked with appropriate run-time libraries for the terminal devices and the computer systems in use.

1.3.2.1 ADP User

Screens must be designed in an interactive mode. The screen design process must be menu driven. The screen designer must:

- Be provided a mechanism for specifying data types, values, and/or range of values that can be entered on a form.
- Be able to position text, data fields, and error messages in any space dimensions supported by the terminal device on which the screen will be used.
- Be able to design forms for fixed and variable length data fields.
- Be able to re-use portions of other screens.

A sample form (master) must be capable of being displayed at the same time that a data entry form is being displayed. The data entered on a form must be saved as an ASCII transaction file. Transaction files generated by data entry must be usable by COBOL programs. The screen designer must be able to edit forms in the database and maintain version control. The application programmer must be able to:

- Provide default values for all user responses.
- Directly control the sequencing of screens.
- Specify menus that are inclusive or exclusive.

1.3.2.2 Functional User

A functional user must:

- Be able to edit within data fields being created but unable to change the pre-defined portions of the form.
- Be able to control the traversal of the screen (e.g., the ability to go back to a previously filled entry for correction).
- Be provided menus that include prompts and help information that is pertinent to his/her form(s) of interest.
- Be able to recall, display and/or print any single form or all forms completed during a session.
- Be warned with an audible sound or visible signal when he/she has exceeded the space allowed for a pre-defined data field and when the transaction space has been completed and a user action is required.

Keystrokes will be minimized for invoking menus or forms, for data entry, and for editing. Uniform menus must be displayed with illogical choices precluded. The cursor will be restricted to valid fields for data entry and menu selection.

1.3.3 Technical Enviornment

The Forms Management System must be designed to operate on VT-100 and IBM 3270 terminals and IBM-PC compatible work stations. Dependencies on operating system facilities will be isolated in modules that are part of the run-time library for an application program. The Forms ManagementSystem must be capable of exploiting a color monitor. Modifications to screen appearance should not require a change to the application program that uses that screen.

2.0 OVERVIEW OF THE FORMS MANAGEMENT SYSTEM DESIGN AND APPLICATION INTERFACE

The Forms Management System (FMS) is composed of a set of Ada programs for the design and execution of forms in applications programs. The execution of a form may entail the construction of a data file containing the responses required by a form or sequence of forms. Section 2.1 provides definitions for the terms used in this document.

2.1 DEFINITION OF TERMS

Section 2.1 defines the terms used in the remainder of this document. These terms reflect the DLA programmer and user environment and the concepts used in the Forms Management System.

Application: The programs and forms required to carry out some task or set of tasks. An application program contains the executable code. The forms contain the display screens. Note that if an application performs several tasks, then that application may require several forms. The forms associated with the application are maintained in a Forms Library.

Applications Programmer: The DSAC personnel responsible for designing and developing an application. Although many individuals may be involved, the term refers to the collective group.

Archival Forms Library: The central repository of all form definitions. They will most likely be located at DSAC. This library includes previous versions of forms.

Box: A box is a fundamental structure in the Forms Management System. There are four box types: containing, text, entry, and menu.

Containing Box: The Forms Management System organizes the components of a form into a hierarchy. Forms may contain a number of screens. Screens may contain a number of components. In addition, components may be grouped. This is done with a containing box. A containing box may also contain several screen components and may be contained within other containing boxes. This containment of boxes within other boxes describes a box hierarchy. In fact, the entire form is defined in terms of a containing box for the entire form, which contains a containing box for each screen, etc. The advantage of this approach is that logically connected boxes may be grouped together.

Data Verification: The Screen Manager provides a limited verification capability for entry box. An entry box may have a data type (numerical, monetary, alphabetic (letters only), and free text). Thus, a letter entered into a numerical field would cause an audible beep. The letter would not be entered into the form and the cursor would not move. In addition, numerical and monetary fields may have an associated range. Alphabetic and free text fields may have a mask. The mask provides a character by character validation of input. The significant mask characters are:

space	: allows any character
'9'	: allows only digits
U'	: allows only upper case letters
'l'	: allows only lower case letters
'a'	: allows any letters
other	: requires the specified digit

For example, a date field might have an associated mask of: "99/99/9999". The 9s require digits and the slashes require slashes.

Entry Box: A component of a screen which associates a prompt field and an entry field. The prompt field should tell the functional user what input is required. The entry field is provided for the functional user to input data.

Form: A sequentially connected group of screens associated with a specific task. Operationally, this means that the Screen Manager processes only one form at a time. The applications program must invoke the Screen Manager explicitly for each form in an application. The Screen Manager will display each screen associated with the form in turn, retrieving user input for each response box within each screen.

Form Invocation: Form invocation entails the display of all static text and the retrieval of any required user responses. Since forms may have many screens, this process takes place one screen at a time. All static text (prompts, titles, etc.) are displayed. Then for each response field (i.e., entry and menu boxes), the cursor is placed in the response area, the user provides the response and presses the ENTER key, and the cursor moves to the next response field. This repeats until the last response field is processed. Then the next screen is processed until the entire form is completed.

Form Load: Before the Screen Manager can be called to invoke a form, the form must be loaded. This involves retrieving the static description of the form from the Forms Library and the construction of the internal data structures describing the dynamic behavior of the form. This includes allocating space for input, setting up default values, etc.

Functional User: The ultimate end user of an application. The functional user is assigned some task, such as processing a batch of order forms, that requires the execution of an application.

Help Box: Each box in a form may have an associated Help Box. The purpose of the help box is to provide guidance to the functional user on what is expected. A help box might contain a sample entry for an entry box, or a more complete explanation of the choices in a menu box. Help boxes are actually special purpose pop-up boxes. They are invoked when the use presses a Help function key. When this occurs, the help box associated with the current entry box (i.e., the box in which the cursor is located) is invoked. If the current entry box does not have a help box, then the entry box's containing box is searched for its help box. This continues up the box hierarchy until a help box is found or until the top of the hierarchy.

Keyword: A keyword is used to help identify and retrieve Forms. For example, a form for ordering shoes could have the associate key words: order, shoes, purchase.

Menu Box: A component of a screen which display a set of items from which the user may make selections. This can be regarded as an alternative to the entry box where the possible input is one (or a few) of a small set of possible values. Note that there is no control information implicit within a menu box. Rather, control must be imposed by the application program. That is, an application program invokes a menu box via the Screen Manager. When the user makes a selection, control is returned to the application program. Then the GET_SELECTION function is called to determine the user's choice. At that time, the applications program may make control decisions based on the value of the menu choice. Also note that Entry and Menu boxes may be freely mixed on a screen.

Operational Forms Library: The set of forms sent to an operational site. This library contains only the most current form definitions and only those forms needed at a given site.

Pop-up Box: The Forms Management System allows the definition of pop-up boxes. A pop-up box may be associated with a given screen, but it is not displayed when the rest of the screen is displayed. This is because a pop-up box, by its very nature, pops-up, interrupting the normal flow of the screen, and disappears after its purpose is completed. For example, help displays may be defined in pop-up boxes. When a user presses a Help key, a display will pop-up explaining what is expected.

Program Module: Within the Forms Management System, a program module (or simply module) is an Ada package or subprogram.

Response Box: Either a menu box or an entry box. That is any box requiring a user response.

Screen Designer: The DSAC personnel responsible for designing the forms and recording those forms in the Archival Forms Library and for generating the Operational Forms Library. The screen designer will probably also be the applications programmer.

Screen: A set of boxes that are displayed simultaneously on the user's terminal. The boxes may be of four types: Containing, Text, Entry, and Menu.

Screen Component: A screen component (or simply component) is the smallest addressable unit in the Forms Management System. A component is a text box, an entry box, or a menu box. Addressability refers to the fact that the screen manager can be used to invoke an individual box, a collection of boxes grouped within a containing box, an entire screen, or an entire form composed of several screens.

Task: The functional users perform tasks requiring the use of applications programs and the processing of forms. Examples of tasks are processing a batch of equipment order forms, processing a message, and initiating a database query.

2.2 FORMS MANAGEMENT SYSTEM

There are five major program modules within FMS: Forms_Editor, Screen Manager, Screen_Controller, Screen_Printer, Forms_Library. The Forms Editor is a stand-alone program for generating forms. In addition, the Forms Editor provides mechanisms for searching the Forms Library and generating an operational library from the archival library. Section 2.3 describes the conceptual structure that underlies the form design and definition process. Table 1 lists the commands available in the Forms Editor organized into command groups.

The Forms Librarian controls the database of form definitions. Form definitions describe the appearance and required end user responses. Forms may require several screens of display and input. The Forms Librarian adds new forms and extracts existing forms under the control of the Forms Editor.

Table 1. Forms Editor Command Groups

Main Command Groups:

FILE	: File handling commands
BOX	: Box handling commands
DISPLAY	: Display attribute commands
ENTRY	: Entry box commands
MENU	: Menu box commands

FILE handling commands:

NEW	: Create a new form.
OPEN	: Open an existing form.
SAVE	: Save the form.
SAVE AS	: Save the form under a new name.
PRINT	: Print the form.
PRINT ALL	: Print all forms in the library.
PURGE	: Purge the forms library.
QUIT	: Exit the Forms Editor.
LIST	: List the forms in the library.

Table 1. Forms Editor Command Groups (Continued)

BOX handling commands:

CREATE	: Create a box within the current containing box.
DELETE	: Delete a box and any subordinate boxes.
REPOSITION	: Change a box's position on the screen.
RESIZE	: Change the box's size on the screen.
LIST BOXES	: List and select boxes
HELP	: Define a help box for the current box.

DISPLAY attribute commands:

FOREGROUND	: Set the foreground attribute.
BACKGROUND	: Set the background attribute.
BORDER	: Set the border visibility.
TEXT POSITION	: Set the position of any text.
TEXT SIZE	: Set the size of the text area.
TEXT CONTENT	: Set the value of the text.

ENTRY box commands:

ENTRY FIELD POSITION	: Set entry field screen position.
ENTRY FIELD SIZE	: Set entry field buffer size.
ENTRY FIELD DEFAULT	: Set a default value.
ENTRY FIELD ATTRIBUTE	: Set a display attribute.
ENTRY FIELD TYPE	: Set the data type.
ENTRY FIELD MASK	: Set the mask for alpha data.
ENTRY FIELD RANGE	: Set the range for numeric data.

MENU box commands:

ADD ITEM	: Add an item to the menu.
DELETE ITEM	: Delete an item from the menu.
SELECTION TYPE	: Toggle exclusive vs. inclusive.
DEFAULT ITEM	: Toggle item as selected or not.
AVAILABLE ITEM	: Toggle item as available or not.
AVAILABLE ATTRIBUTE	: Set available item display attribute.
UNAVAILABLE ATTRIBUTE	: Set unavailable item display attribute.
SELECTED ATTRIBUTE	: Set selected item display attribute.

The Screen Manager displays a specific sequence of boxes on the end user's terminal and retrieves responses. These responses are buffered within the Screen Manager and provided to the calling program on request.

The Screen Controller provides procedures for manipulating a display screen and recording its content. These procedures include:

SAVE_REGION
RESTORE_REGION
SCROLL_REGION
DRAW_BORDER
RETURN_REGION

The Screen Primitives package provides a number of simple procedures conforming to ANSI standard terminal control escape sequences. These include procedures for moving the cursor clearing the screen, obtaining the cursor's position, etc.

Figure 1 is an overview of the FMS and the Application Interface.

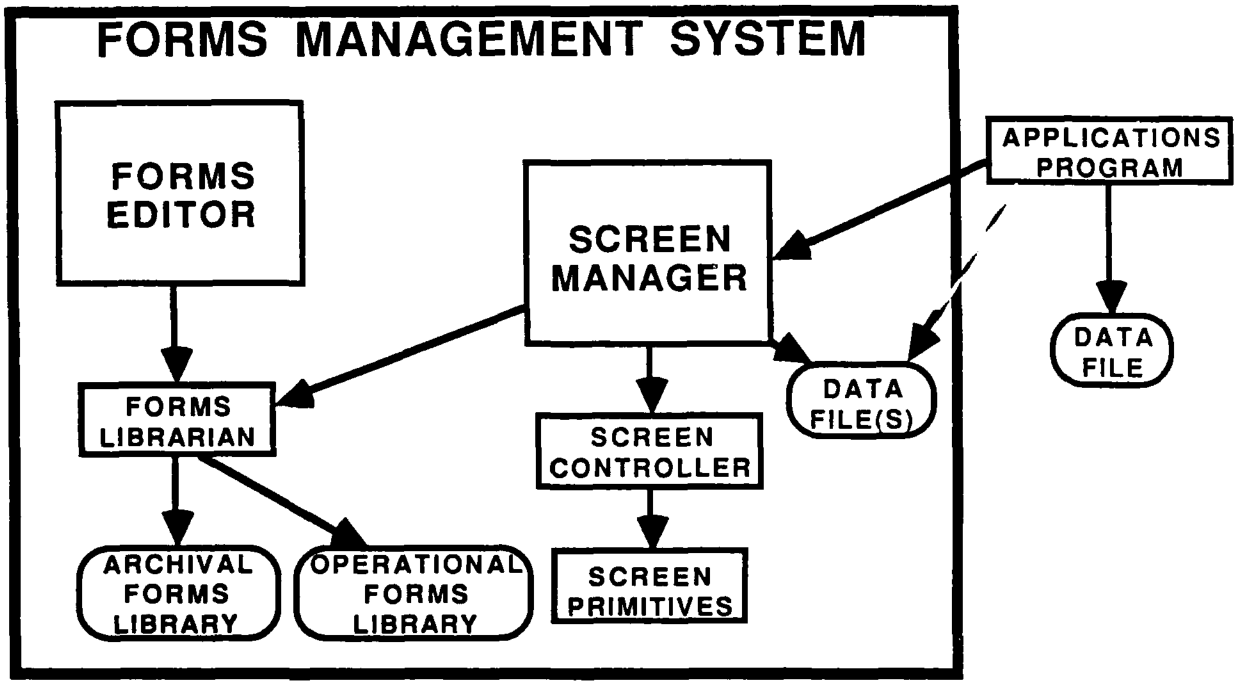

Figure 1. Overview

2.3 CONCEPTUAL STRUCTURE

The following text expands on the major design concepts of the Forms Management System. The major design concepts are:

- The use of boxes as a screen organizational unit;

- The use of screen definitions to promote modularity; and,
- The use of functions keys.

2.3.1 Boxes

The primary model for screen organization is the box. Boxes may be one of four different types: Containing, Text, Entry, or Menu. Containing Boxes, as the name implies, contain other boxes. This is the method by which two or more boxes, e.g. a text box providing instructions and a menu box, may be grouped together. Generally, a screen is defined as one Containing Box and several subordinate boxes of various types. However, Containing Boxes may contain other containing boxes. A box hierarchy is defined as a Containing Box and all its subordinate boxes. Entry and Menu Boxes are collectively referred to as input or response boxes.

All box types have a number of attributes which control their appearance on a screen. The location of a box on a screen is relative to its containing box. Thus, the relocation of a Containing Box will automatically relocate subordinate boxes. A box's border may be visible or invisible. Boxes may have prescribed background and foreground display attributes, such as reverse video or color, although display attributes are limited by the host computer system and terminal display characteristics. Finally, boxes may have one of three invocation actions. The invocation action specifies whether the screen is cleared before the box is displayed, or whether the box overwrites the previous screen contents, or whether the box "pops-up", thus preserving the previous screen contents.

A Text Box contains text, such as screen titles, that are to be displayed but require no response from the end user of the application. All box types may contain text. The explicit use of text boxes allows several messages to appear at different locations on the screen.

Entry Boxes specify a prompt/input pair. The screen designer may specify the location, size and display mode of the prompt and the reply fields. The screen manager system allows the end user to perform line editing of the input and can perform limited data validation. The entry field may be restricted to numeric, monetary, or textual data. Numeric and monetary fields may have a specified range and textual entry fields may have an associated mask. More detailed validation and data conversions must be done by the applications program.

Menu boxes provide a list of options that may be selected. At any one time, some of the options may be available to the user, while others are unavailable. For example, an application may have a menu for file manipulation. The menu may include options for opening a file, saving a file, and printing a file. The applications developer may wish to have all options visible when the box is displayed. However, when the application is first started, only the open option may be available to the user, since the other two options depend on an opened file. The menu box definition may include different display modes for available, unavailable, and selected choices.

Furthermore, menu boxes may be exclusive, meaning that any choice precludes any other available choice from that menu, or inclusive, meaning that several choices may be valid at the same time.

2.3.2 The Screen Definition

The screen definition contains the definitions of one or more screens. As noted previously, a screen is generally represented by a containing box with a number of subordinate boxes. Each box definition includes its name and all information pertinent to that box. Box names need only be unique within the context of the box's containing box. Applications programs identify individual boxes by a "dot" notation. For example, if box "A" is a containing box and box "B" is a menu box within "A", then the menu box is accessed with the name sequence "A.B"

It should be noted that the screen definition represents a starting point for an application. The Screen Manager provides procedures for making modifications to box descriptions. These modifications include the default values of entry boxes, the available items of menu boxes, as well as the menu items themselves.

2.3.3 Function Keys

This document makes references to a number of function keys for controlling the definition and traversal of screens. This document is kept generic by naming the keys according to their function. Specific implementations of the system will bind the function keys to physical keys on the keyboard. For example, cursor movement functions would most likely be bound to an IBM PC keyboard's arrow keys. The following is a list of functions and a suggested binding on an IBM PC keyboard.

Function	IBM PC Binding
NEXT FIELD	TAB or ENTER
PREVIOUS FIELD	shift-TAB
SELECT	ENTER
END OPERATION	END (unshifted 1)
CURSOR_UP	UP ARROW (unshifted 8)
CURSOR_DOWN	DOWN ARROW (unshifted 2)
CURSOR_LEFT	LEFT ARROW (unshifted 4)
CURSOR_RIGHT	RIGHT ARROW (unshifted 6)

The NEXT FIELD and PREVIOUS FIELD functions are used to traverse a sequence of input boxes. The SELECT function is used to select a box in the SCREEN EDITOR system or select an item in a menu in the SCREEN MANAGER. The END OPERATION function is used to terminate some operation such as relocating a box in the SCREEN EDITOR. The cursor control functions are used to move the cursor on the screen.

2.4 SCENARIO FOR APPLICATION PROGRAM INTERFACE

The following scenario illustrates how an applications programmer would use the Screen Manager.

1. The Screen Manager must be "withed" into the body of the applications program.

2. The appropriate form must be loaded by a call to LOAD_FORM. This call will cause the Screen Manager to access the Operational Forms Library (via the Forms Librarian Package) and retrieve the description of the form.

3. The form is invoked by a call to INVOKE_FORM that specifies the form name. Here, the Screen Manager takes control, displaying the screens defined in the form and retrieving the functional users responses. When the form is completed, control is returned to the applications program.

4. After the call to INVOKE_FORM, the applications program issues a series of calls to GET_ENTRY and GET_SELECTION. These calls return the functional user's responses to specified entry and menu boxes.

5. The applications program may then record the responses in a transaction file, access a database, or use the data for program control.

6. Then the form may be invoked again or another form may be invoked for a different operation Figure 2 shows the flow of data in the FMS.

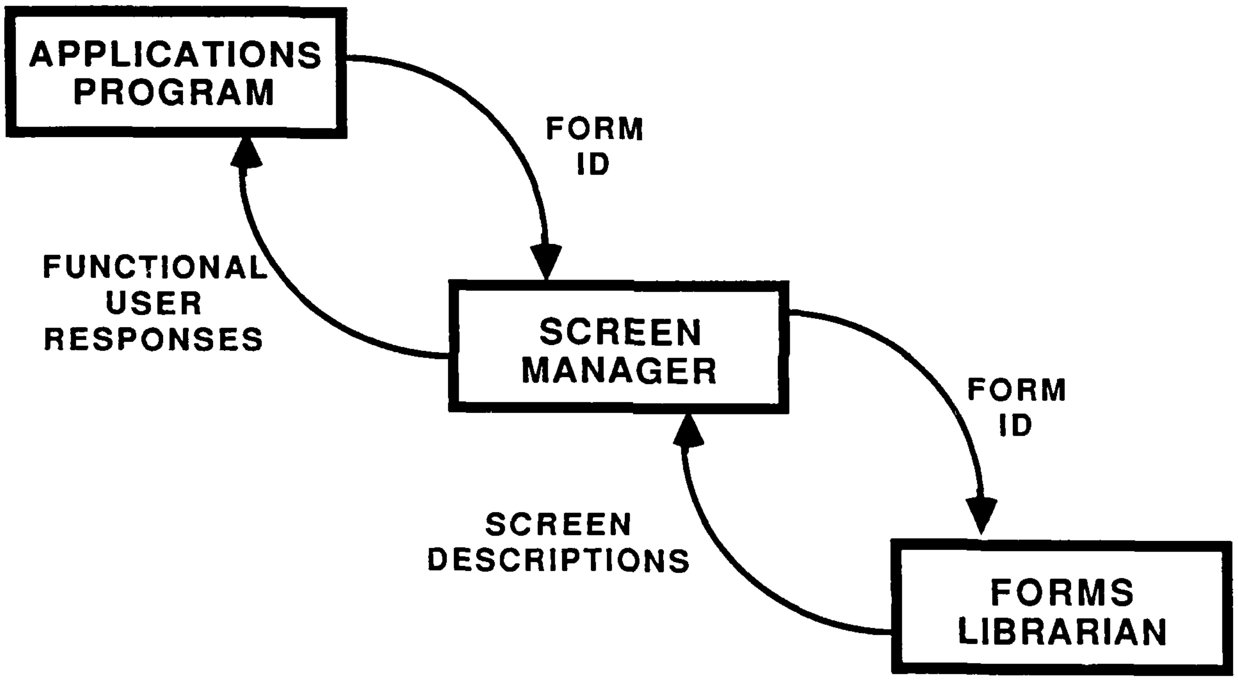

Figure 2. Data Flow

7. The applications programs supplies a Form Name (Form ID) to the Screen Manager which passes it on to the Forms Librarian.

8. The Forms Librarian returns a description of the form to the Screen Manager.

9. The Screen Manager uses this description to retrieve input from the Functional User which is passed back to the Applications Program.

Figure 3 shows the interaction of the principle procedures in the run time support program modules of the FMS. The four major procedures of the Screen manager are LOAD_FORM, GET_ENTRY, GET_SELECTION, and INVOKE_FORM. Although other procedures exist in the Screen Manager, these four represent a minimum set for effectively using the FMS. The LOAD_FORM retrieves a Form Description from the Forms_Librarian via a call to GET_FORM. INVOKE_FORM manipulates the terminal's screen via calls to the Screen Controller procedures. GET_ENTRY and GET_SELECTION operate on data structures within the Screen Manager.

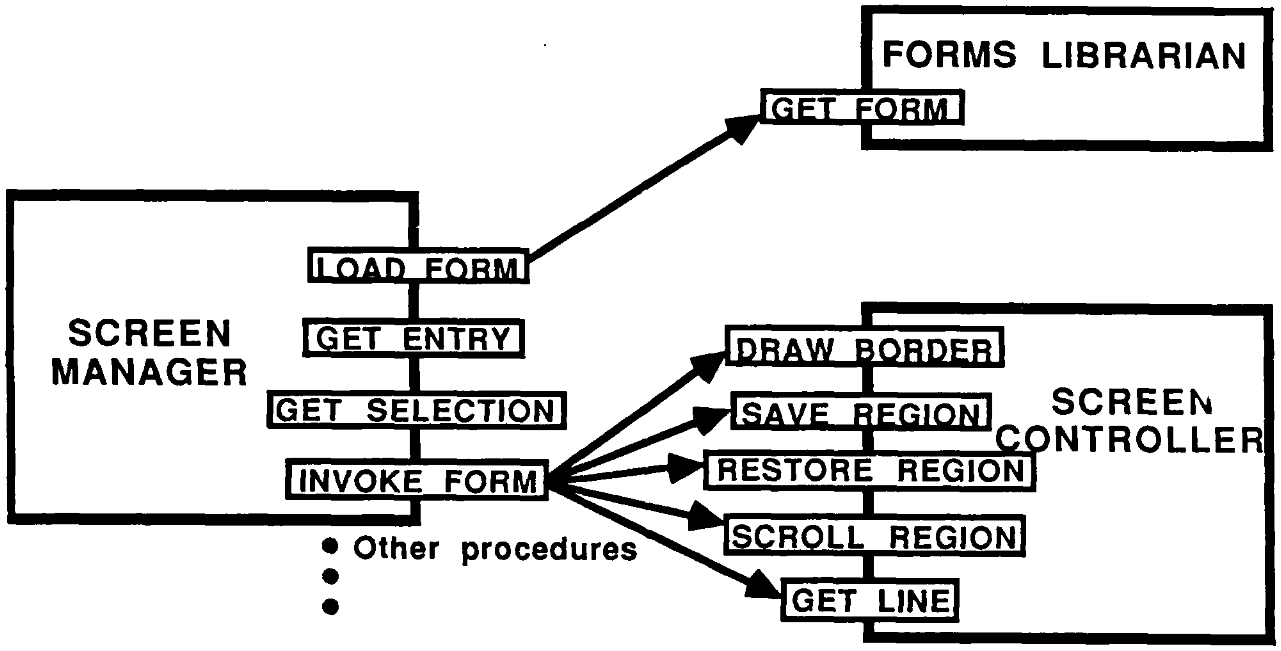

Figure 3. Program Module Interaction

3.0 DESIGN SPECIFICATIONS

3.1 FORMS MANAGEMENT SYSTEM

This design of the Forms Management System (FMS) is specified in an Ada-like program design language.

3.1.1 FORMS_EDITOR

```
---------------------------------------------------------------
-- The FORMS_EDITOR is a stand along program for designing    --
-- and file forms. When the program is started, the           --
-- form designer (i.e., the user) is given the option of      --
-- starting a new form, editing an old form, printing a       --
-- form description, or generating an operation forms         --
-- library from the archival forms library.                   --
--
-- The process of designing a form is largely menu driven.    --
-- Extensive use is made of cursor control to define the      --
-- size and shapes of screen components (boxes) and           --
-- subcomponents (text, entry fields, etc.).                  --
---------------------------------------------------------------

with SCREEN_MANAGER;
with FORMS_LIBRARIAN;
with KEY_VALUES; -- maps function key values.
procedure FORMEDIT is

Package   SM renames SCREEN_MANAGER;
Package   FL renames FORMS_LIBRARIAN;

-- A special menu handler is needed because of the use of a
-- horizontal menu, i.e., the top line of command groups. This
-- special handler will accept horizontal cursor keys and returns
-- the key value. The normal handler only accepts the vertical
-- cursor control keys and the select key.

  procedure SPECIAL_MENU_HANDLER(MENU_BOX_NAME : in STRING;
                                 RETURN_CODE   : out CHARACTER)
                    is separate;
```

```
-- NEXT_COMMAND_SET takes the current command set and the
-- horizontal cursor control key and determines the next
-- command set. For example. if the current command set is
-- "FILE" and the CURSOR_LEFT key were pressed, the current
-- command set would wrap around to the "MENU" command set.

  procedure NEXT_COMMAND_SET(CURRENT_COMMAND_SET : in out STRING;
                             CURSOR_DIRECTION    : in CHARACTER)
                             is separate;

-- PROCESS_COMMAND is the procedure that actually performs the work.
-- Note that both the command and its command set is given to the
-- procedure so that there is no confusion if the same command string
-- appears in more than one command set.

  procedure PROCESS_COMMAND(COMMAND_SET : in STRING;
                              COMMAND   : in STRING)
                             is separate;

  CURRENT_COMMAND_SET : STRING[1..7];
  CURRENT_COMMAND     : STRING[1..30];
  MENU_RETURN_CODE    : CHARACTER;

begin

-- First, load the screens associated with the
-- For   ditor program.

  SM.LOAD_FORM("Forms_Editor");

-- Next, invoke the first screen (which merely clears the screen)

  SM.INVOKE_BOX("Forms_Editor.Clear_Screen");

-- Now, setup the FILE command set as the current command
-- set. The current command is set to OPEN, although this
-- will be changed when the FILE command box is invoked.

  CURRENT_COMMAND_SET := "FILE";
  CURRENT_COMMAND     := "OPEN";

-- Now, loop until the screen designer wishes to quit
-- Note that we are assuming the existence of pop-up box
-- definitions for "Forms_Editor.FILE", "Forms_Editor.DISPLAY",
-- etc.
```

```
loop
          SPECIAL_MENU_HANDLER("Forms_Editor." &
         CURRENT_COMMAND_SET,
                                   MENU_RETURN_CODE);

   if MENU_RETURN_CODE = KEY_VALUES.SELECT then
          begin
            CURRENT_COMMAND := SM.GET_ENTRY("Forms_Editor." &
                                   CURRENT_COMMAND_SET);

-- Note, we might want to verify before actually quitting

            exit when CURRENT_COMMAND = "QUIT";
            PROCESS_COMMAND(CURRENT_COMMAND_SET,
            CURRENT_COMMAND);
          end;

          else
            NEXT_COMMAND(CURRENT_COMMAND_SET,
            MENU_RETURN_CODE);
  end loop;

end FORM EDIT;   -- The procedure
```

3.1.2 SCREEN_MANAGER

```
package SCREEN_MANAGER is
  -----------------------------------------------------------------
    --The SCREEN_MANAGER is the primary run-time interface into        --
    --the Forms Management System. Typically, the SCREEN_MANAGER is    --
    --used in the following sequence:                                  --
    --                                                                 --
    -- 1. Load a form - LOAD_FORM                                      --
    -- 2. Invoke a form - INVOKE_FORM                                  --
    -- 3. Retrieve end user responses - GET_ENTRY or GET_SELECTION     --
    --                                                                 --
    -- Variations of use:                                              --
    -- * Steps 2 and 3 may be done iteratively for applications        --
    --   in which a batch of forms are being processed.                --
    --                                                                 --
    -- * The application could take advantage of the data file         --
    --   capabilities of the SCREEN_MANGER. In this case, Step 1       --
    --   would also have to open a data file (OPEN_DATA_FILE)          --
    --   and Step 3 would be changed to store the input data into      --
    --   the data file (STORE_DATA).                                   --
    --                                                                 --
                                                                       --
    -- * If the application provided data verification, then          --
    --   additional steps could be provided to test the data,         --
    --   display error messages, and request corrected data.          --
    --                                                                 --
  -------------------------------------------------------------------------

-- The LOAD_FORM procedure retrieves the specified from from
--           the Forms Library. If the form does not exist, then the
--           exception UNKNOWN_FORM is raised.

procedure LOAD_FORM(FORM_NAME : in STRING);

-- The OPEN_DATA_FILE procedure is used if the applications
-- program wishes the SCREEN_MANAGER to handle file buffering
-- of the input data. Details of how the information is stored
-- are described in the WRITE_DATA procedure description.

procedure OPEN_DATA_FILE(FILE_NAME : in STRING;
                         DATA_FILE : in out FILE_TYPE);

-- CLOSE_DATA_FILE is the companion procedure for OPEN_DATA_FILE.

procedure CLOSE_DATA_FILE(DATA_FILE : in out FILE_TYPE);

-- The INVOKE_FORM procedure displays the static contents of
-- a form and retrieves the end user's responses. These responses
-- are buffered for subsequent retrieval by the application
-- program via the GET_ENTRY or GET_SELECTION function or by
-- calling, STORE_DATA. If a form is composed of more than one
-- screen, then each screen is invoked in turn.
-- INVOKE_FORM raises an UNKNOWN_FORM exception if the form has
-- not yet been loaded.

procedure INVOKE_FORM(FORM_NAME : in STRING);

-- The INVOKE_BOX procedure is used to invoke parts of a form.
-- For example, INVOKE_BOX could be used to display an error
-- message, or a special entry field not normally displayed
-- with the rest of the form, or the application program may
-- need to provide more interaction than simply displaying
-- a static form and retrieving the responses.
-- INVOKE_BOX raises an UNKNOWN_BOX exception if the box cannot
-- be found within the currently loaded form.

procedure INVOKE_BOX(BOX_NAME : in STRING);

-- The WRITE_DATA procedure takes the current contents of the
-- form buffers and writes them to the currently opened data
-- file. Each record is essentially a concatenation of all of the
-- user response fields in the form. Responses to menus are
-- represented by the string displayed for the selected menu
-- item. Individual entries are separated by a field separator
-- code (ASCII.FS or decimal 28). In addition, trailing spaces
-- are deleted.

procedure WRITE_DATA(DATA_FILE : in FILE_TYPE);

-- The READ_DATA procedure provides a mechanism for retrieving
-- previously entered data. The returned VALUE is the contents of
-- the data record. That is, only the user responses are included
-- and each response is separated by a field separator code
-- (ASCII.FS or decimal 28). This string can be manipulated by
-- the application program or provided as an argument to the
-- DISPLAY_DATA procedure to display the data on the user's
-- terminal.

procedure READ_DATA(DATA_FILE          : in FILE_TYPE;
                    TRANSACTION_NUMBER : in NATURAL;
                    VALUE              : out STRING);

-- The DELETE_DATA removes a transaction from the data file

procedure DELETE_DATA(DATA_FILE          : in FILE_TYPE;
                      TRANSACTION_NUMBER : in NATURAL);

-- The DISPLAY_DATA procedure takes a string (retrieved from a
-- data file by the READ_DATA procedure) and displays the string
-- in a form on the user's terminal. This is done by setting
-- the individual data fields as default values for each of the
-- response fields and invoking the form. It should be noted
-- that there could be unexpected results if the applications
-- program changed the current form (by calling LOAD_FORM)
-- after the original data was saved in the data file and before
-- the data is redisplayed on the screen. If possible,
-- DISPLAY_DATA will detect any such changes and raise the
-- INAPPROPRIATE_ACTION exception. (Note that such a change
-- would be undetectable if the number, types, and sizes of the
-- user response fields were the same in two different forms.)

procedure DISPLAY_DATA(VALUE : in STRING);

-- The GET_ENTRY procedure is used to retrieve response values to
-- specific user response boxes. If the referenced box is an
-- entry box, then the returned value is the text entered by the
-- user (or its default value). If the referenced box is not an
-- entry box, then the exception INAPPROPRIATE_ACTION is raised.
-- If the referenced box can not be found, then the UNKNOWN_BOX
-- exception is raised.

function GET_ENTRY(BOX_NAME : in STRING) return STRING;

-- The GET_SELECTION procedure is used to retrieve the selected
-- choices from a menu. Since a menu may be inclusive (i.e., more
-- than one item may be selected), the return value is a pointer
-- to a chain of selected items. Each entry in the chain contains
-- a string representing a selected item and a pointer to the
-- next item in the string. This structure is defined in the
-- following data types:

type MENU_LIST_TYPE;

type MENU_LIST_ACCESS_TYPE is access MENU_LIST_TYPE;

type MENU_LIST_TYPE is
            record
               ITEM_NAME : STRING;
               NEXT           : MENU_LIST_ACCESS_TYPE;
            end record;

-- The GET_SELECTION function returns the selected items from a
-- menu box. If the referenced box does not exist, then the
-- UNKNOWN_BOX exception is raised. If the referenced box is not
-- a menu box, then the INAPPROPRIATE_ACTION exception is raised.

function GET_SELECTION(BOX_NAME: in STRING)
                        return MENU_LIST_ACCESS_TYPE;

-- It is customary to always display menus in a consistent
-- fashion. That is, all possible choices in a menu are
-- displayed whenever the menu is displayed. However, at any
-- given time, some choices may be inappropriate. For example,
-- an application that manipulates files may have a menu that
-- contains choices for opening, closing, and printing a given
-- file. When the application is started, only the open choice
-- is appropriate. After a file is opened, close and print become
-- appropriate, but open is no longer appropriate. The
-- SCREEN_MANAGER provides for this situation with the following
-- pair of procedures. If the referenced box cannot be found in
-- the current form, then the UNKNOWN_BOX exception is raised. If
-- the referenced box is not a menu box, or if the menu box cannot be found, then the
-- INAPPROPRIATE_ACTION exception is raised. No errors are raised
-- if the procedures have no effect on the menu (e.g., calling
-- MAKE_AVAILABLE and referencing an item that is already
--  available.
```

```
procedure MAKE_AVAILABLE(MENU_BOX_NAME : in STRING;
                         MENU_ITEM     : in STRING);

procedure MAKE_UNAVAILABLE(MENU_BOX_NAME : in STRING;
                           MENU_ITEM     : in STRING);

-- Some applications may have to build menus at run time. That
-- is, the contents of a menu may depend on data that is only
-- available at run time. The following two procedures are used
-- to add and delete items to a menu. The exceptions UNKNOWN_BOX
-- and INAPPROPRIATE_ACTION are raised for erroneous calls.

procedure ADD_ITEM(MENU_BOX_NAME : in STRING;
                   NEW_ITEM      : in STRING);

procedure DELETE_ITEM(MENU_BOX_NAME : in STRING;
                      OLD_ITEM      : in STRING);

UNKNOWN_FORM            : exception;
UNKNOWN_BOX             : exception;
INAPPROPRIATE_ACTION    : exception;

end SCREEN_MANAGER;
```

3.1.3 SCREEN_CONTROLLER

```
package SCREEN_CONTROLLER is
  --------------------------------------------------------
  --The SCREEN_CONTROLLER package provides the following   --
  -- capabilities:                                        --
  --                                                      --
  -- 1. Aggregate screen control functions such as drawing --
  --    a box.                                            --
  --                                                      --
  -- 2. An internal screen image that mimics the terminal --
  --    display screen.                                   --
  -- 3. Screen I/O procedures (e.g. GET and PUT) that      --
  --          duplicates screen TEXT_IO while recording the --
  --          activity on the internal screen image.      --
  -- 4. All SCREEN_PRIMITIVES actions which modify the     --
  --          the screen so that those changes can be recorded --
  --    on the internal screen.                           --
  --------------------------------------------------------

-- The DRAW_BOX procedure draws a box with a character defined
--          in the DRAW_BOX procedure body. The arguments describe the
--          upper left hand and lower right hand corner of the box.
```

```
procedure DRAW_BOX(TOP, LEFT, BOTTOM, RIGHT : in NATURAL);

-- The SAVE_REGION and RESTORE_REGION assist in the display of
-- pop-up boxes. SAVE_REGION takes the contents of a specified
-- region of the screen and saves it on a stack. RESTORE_REGION
-- takes the region on the top of the stack and displays it on
-- the screen. The use of a stack allows the use of overlapping
-- pop-up boxes. Note that RESTORE_REGION does not need arguments
-- since the region boundaries are stored with the region's
-- image on the stack.

procedure SAVE_REGION(TOP, LEFT, BOTTOM, RIGHT : in NATURAL);
procedure RESTORE_REGION;

-- SCROLL_REGION scrolls the contents of a specified region of
-- the screen. When the argument specifies scrolling up, the top
-- line in the region is deleted, all other lines are moved up
-- one, the bottom line is cleared, and the cursor is placed in
-- the bottom left hand corner of the region. The action is
-- similar when scrolling down, except that the lines are moved
-- down and the cursor is placed in the top left hand corner.

procedure SCROLL_REGION(TOP, LEFT, BOTTOM, RIGHT : in NATURAL;
                        DIRECTION : in (UP, DOWN) := UP);

-- The following procedures mimic the SCREEN_PRIMITIVES
-- procedures except that they record any screen modifications on
-- the internal screen image before calling SCREEN_PRIMITIVES.

procedure HOME_CURSOR;
procedure MOVE_CURSOR_TO(ROW, COLUMN : in NATURAL);
procedure MOVE_CURSOR_UP     (COUNT : in NATURAL := 1);
procedure MOVE_CURSOR_DOWN   (COUNT : in NATURAL := 1);
procedure MOVE_CURSOR_RIGHT  (COUNT : in NATURAL := 1);
procedure MOVE_CURSOR_LEFT   (COUNT : in NATURAL := 1);
procedure CLEAR_SCREEN;
procedure ERASE_END_OF_LINE;

-- The following procedures mimic TEXT_IO procedures for writing
-- to and reading from the user's terminal. They are provided
-- here because any such actions must be reflected in the
-- internal screen representation.

procedure GET            (ITEM : out CHARACTER);
procedure GET            (ITEM : out STRING);
procedure PUT          (ITEM : in CHARACTER);
procedure PUT          (ITEM : out STRING);

end SCREEN_CONTROLLER;
```

3.1.4 SCREEN_PRIMITIVES

```
package SCREEN_PRIMITIVES is

  ------------------------------------------------------------------
  --      This package provides ANSI primitive screen control routines  --
  --      for cursor movement, position reporting, screen clearing,      --
  --      etc. Although the procedures conform to ANSI escape            --
  --      sequences, non-ANSI terminals may be used by providing         --
  --      whatever control strings are necessary.                        --
  ------------------------------------------------------------------

-- HOME_CURSOR moves the cursor to the top left corner of the
-- screen.
procedure HOME_CURSOR;

-- MOVE_CURSOR_TO places the cursor in a specified row and column
-- position.
procedure MOVE_CURSOR_TO(ROW, COLUMN : in NATURAL);

-- MOVE_CURSOR_xx moves the cursor a specified number or lines
-- (rows) in the indicated direction.
-- The default for all relative cursor movement procedures is 1
--  unit.

procedure MOVE_CURSOR_UP     (COUNT : in NATURAL := 1);
procedure MOVE_CURSOR_DOWN   (COUNT : in NATURAL := 1);
procedure MOVE_CURSOR_RIGHT  (COUNT : in NATURAL := 1);
procedure MOVE_CURSOR_LEFT   (COUNT : in NATURAL := 1);

-- REPORT_CURSOR_POSITION returns the current cursor location
procedure REPORT_CURSOR_POSITION(ROW, COLUMN : out NATURAL);

-- CLEAR_SCREEN clears the entire contents of the screen and
-- places the cursor in the top left corner of the terminal.
procedure CLEAR_SCREEN;

-- ERASE_END_OF_LINE erases all text from the current cursor
-- position to the right hand side of the screen.
procedure ERASE_END_OF_LINE;
```

```
-- SQUAK makes an audible signal from the terminal.
 procedure SQUAK;
 end SCREEN_PRIMITIVES;
```

3.1.5 FORMS_LIBRARIAN

```
with FORM_DATA_STRUCTURE;
FDS renames FORM_DATA_STRUCTURE;

package FORMS_LIBRARIAN is

       -------------------------------------------------------------
     --                                                            --
     -- The FORMS_LIBRARIAN is responsible for maintaining the     --
     --  Forms Library. Operations on the library are :            --
     --                                                            --
     --                                                            --
     --          STORE_FORM        - Store a form in the library   --
     --          GET_FORM          - Get the most recent version   --
     --          DELETE_FORM       - Delete the most recent version --
     --          GET_FORMS_LIST    - Get a list of current form names --
     --          PURGE_FORM        - Delete all but most recent version --
     --          ADD_KEYWORD       - Add a keyword to a form       --
     --          DELETE_KEYWORD    - Remove a keyword from a form  --
     --                                                            --
     --                                                            --
     --  The Forms Library is actually a database in which forms   --
     --  are the data objects. Each form has a unique identifier (Form ID). --
     --  In addition, each form has a set of keywords that may be  --
     --  used to identify a form.                                  --
     --                                                            --
     --  The Forms Library is maintained in a file whose name is   --
     --  fixed in the body of the FORMS_LIBRARIAN.                 --
       -------------------------------------------------------------

-- The FORMS_NAME_LIST data structure is used to form a linked
-- list of form names. Each item in the name list is a structure
-- containing the form name and a link to a list of key words.

type KEYWORD_ITEM;

type KEYWORD_ITEM_ACCESS_TYPE is access KEYWORD_ITEM;

type KEYWORD_ITEM is
          record
             KEYWORD : STRING;
             NEXT         : KEYWORD_ITEM_ACCESS_TYPE;
          end record;
```

```
type FORMS_ITEM;

type FORMS_ITEM_ACCESS_TYPE is access FORMS_ITEM;

type FORMS_ITEM is
        record
          FORM_NAME    : STRING;
          KEYWORD_LIST : KEYWORD_ITEM_ACCESS_TYPE;
          NEXT         : FORMS_ITEM_ACCESS_TYPE;
        end record;

-- The STORE_FORM procedure takes a form data structure and
-- stores the form in the library. It is possible for two forms
-- to have the same name. In this case, the older form (i.e., the
-- form already in the library) is assumed to be a previous
-- version. Previous versions are maintained until a PURGE
-- operation is performed.

procedure STORE_FORM(FORM_NAME : in STRING;
                     FORM_STRUCTURE : in FDS.FORM_ACCESS_TYPE);

-- The GET_FORM procedure retrieves a specified form from the
-- library. If the form is not found, then the exception
-- UNKNOWN_FORM is raised.

procedure GET_FORM(FORM_NAME : in STRING;
                   FORM_STRUCTURE : out FDS.FORM_ACCESS_TYPE);

-- The DELETE_FORM procedure removes the most recent version of
-- a specified form from the library. If the form does not exist,
-- the exception UNKNOWN_FORM is raised. If more than one form
-- existed before the delete procedure is called, the second most
-- recent form becomes the current version.

procedure DELETE_FORM(FORM_NAME : in STRING);

-- The GET_FORMS_LIST procedure search is for all forms with
-- keywords matching the (optional) list given in the argument
-- of the procedure.

procedure GET_FORMS_LIST(FORM_NAMES : out FORMS_ITEM_ACCESS_TYPE;
                         KEYWORDS : in KEYWORD_ITEM_ACCESS_TYPE
                                    := null);

-- The PURGE_FORM procedure is used to remove all previous
-- versions of a form. If only one version of a form resides
-- in the library, no action is taken. If no forms of the
-- given name reside in the library, the exception
-- UNKNOWN_FORM is raised.

procedure PURGE_FORM(FORM_NAME : in STRING);

-- The ADD_KEYWORD and DELETE_KEYWORD procedures are used to
-- modify the keywords associated with a form. Only the most
-- recent version of a form is affected. If the form cannot
-- be found, then the exception UNKNOWN_FORM is raised. If
-- an add operation is performed on a form that already has
-- the indicated keyword, or a delete operation is perform
-- on a form that does not have the indicated keyword, no
-- action is taken.

procedure ADD_KEYWORD(FORM_NAME : in STRING;
                      KEYWORD   : in STRING);

procedure DELETE_KEYWORD(FORM_NAME : in STRING;
                         KEYWORD   : in STRING);

UNKNOWN_FORM : exception;

end FORMS_LIBRARIAN;
```

3.2 TYPICAL APPLICATION PROGRAM EXAMPLE

The following is a skeleton program for an applications program using the Screen Manager package.

```
        with SCREEN_MANAGER;
        procedure APS1 is

        SM renames SCREEN_MANAGER;

EMPLOYEE_NAME  is STRING[1 . . 26];
EMPLOYEE_AGE   is NATURAL;

begin
             SM.LOAD_FORM("B5006G"); -- setup so :e arbitrary form
loop

   ----------------------------------------------------------
--Any changes to the form would go here.  Possible changes would be changing default  --
--values, enabling or disabling menu items, etc.                                      --
   ----------------------------------------------------------
```

```
          SM.INVOKE_FORM("B5006G); -- Screen Manager takes control
                                   -- and retrieves user input

    ------------------------------------------------------------------------
--Now we have to retrieve the information from the screen manager. Note that we need --
--to convert AGE from string.                                                       --
    ------------------------------------------------------------------------
          EMPLOYEE_NAME  := SM.GET_ENTRY("B5006G.SCRN1.NAME");
          EMPLOYEE_AGE   := STRING_TO_NUMBER(
                               SM.GET_ENTRY("B5006G.SCRN1.AGE"));

          put (EMPLOYEE_NAME);
          put_line (EMPLOYEE_AGE);

      end loop

    -----------------------------------------------------------
-- Now we can do whatever with the the name and age values.    --
    -----------------------------------------------------------

end.
```

APPENDIX A

ALLOCATION OF REQUIREMENTS TO DESIGN

The following text discusses the system requirement and how each requirement is fulfilled in the design.

High Level Requirements

The system must be a set of Ada packages that are compilable into an Ada program library.

- The design assumes the Ada programming language as the implementation language. The Program Design Language (PDL) will use Ada constructs to describe the program designs.

The system must be usable for Ada demonstration projects and adaptable to a prototype Ada application;

- The Forms Management System provides a well-defined solution to an existing need of DSAC. The scope of the system is sufficient to demonstrate most of the capabilities of the Ada programming language (with the possible exception of tasking). The system is sufficiently complex so as to provide an interesting demonstration.

The system must provide functions to create, delete, and modify menu-type screens.

- Creation, deletion, and modification facilities are provided in the Forms Editor. The Screen Manager is a run-time support package that is used to manipulate the forms. Menus are a type of user response box that is supported by all program modules in the Forms Management System.

The system must provide functions to invoke menu-type screen from applications.

- This requirement is satisfied by the Screen Manager.

The system must provide functions to create, delete, and modify form-type screens.

- The Forms Editor provides facilities to create, delete, and modify forms. The Screen Manager provides facilities for manipulating data entry forms.

The system must provide functions to gather data input from form-type screens and to pass this data, through a defined interface, to application programs written in Ada or in COBOL.

- The requirement is satisfied by the Screen Manager. The Screen Manager includes a callable procedure for writing the variable content of a screen (i.e. the data entered by the functional user) to a data file. The data is organized as one form per record with individual data fields separated by a field separator character (ASCII.FS).

The system must perform a limited but critical amount of input data verification. This verification must be performed with the same rigor for all terminal types (i.e., application programs must be assured of the same verification, or lack or it, without knowing the terminal type in use).

- This requirement is satisfied in the form description and the Screen Manager. Specifically, the form description includes a type for data entry boxes (e.g., numeric, monetary, alphanumeric) and either a range for numeric fields or a mask for alphabetic fields. The Screen Manager compares end user input to the entry field. Although the specific mechanism by which verification will be done will depend on the terminal type (esp. the 3270), the mechanism will be transparent to the application program. This may result in minor differences in behavior for the functional user depending on the terminal type. Specifically, 3270 type terminals may have to delay verification until an entire field is completed, while other terminals could allow character by character verification.

The system must be time-efficient such that the functional user sees no degradation of performance from the currently available menu programs.

- With two exceptions, all modules have an execution complexity of O(n). The exceptions are the retrieval of specific forms in the forms library and access of specific boxes in a forms definition. The forms library is indexed, thus minimizing the time needed to search for a specific form. The screen manager keeps a specific form definition in a tree structure, thus reducing the time needed to access a specific component. The anticipated complexity of specific forms should not noticeably effect execution speed.

The system must be designed so that all or portions of it can be developed by the programmers of DSAC to demonstrate their skills in software engineering and the Ada language.

- This requirement is partially satisfied by the design approach, and a specification that can be refined by DSAC. (Note: The low level terminal control packages will require specialized skill/knowledge).

Lower Level Requirements

Forms must be designed in an interactive mode.

- This requirement is satisfied by the Forms Editor.

The form design process must be menu driven.

- This requirement is satisfied by the Forms Editor.

The screen designer must:

- be provided a mechanism for specifying data type, values, and/or range of values that can be entered on the form.
- be able to position text, data field, and error messages in any space dimensions supported by the terminal device on which the form will be used.
- be able to design forms for fixed and variable length data fields.
- be able to re-use portions of other forms.

- This requirement is satisfied by the Forms Editor. Data types, values, and ranges are attributes of entry boxes that are explicitly modifiable by the screen designer. The position of various screen components are also explicitly modifiable. Currently, the design fixes the maximum length of entries to the size of the entry field. However, the entry field could be arbitrarily large, up to the size of the terminal's display screen. If this is not sufficient, then the current design can easily be modified to include a flag in the entry box description to indication a fixed or variable length entry. However, even variable length entries must have an indication of maximum size.

A sample form (master) must be capable of being displayed at the same time that a data entry form is being displayed.

- The screen design may include text boxes displaying sample inputs next to data entry boxes. In addition, the screen designer can associate help screens with each entry box on a form. The help box could contain sample input when the functional user presses the HELP function key.

The data entered on the form must be saved as on ASCII transaction file.

- This requirement is satisfied by the data file capabilities of the Screen Manager.

Transaction files generated by data entry must be usable by COBOL programs.

- It is assumed that COBOL programs can read ASCII files. In addition, the Forms Editor can generate a field by field description of the data file generated by the Forms Processor. The syntax of this description has not yet been decided, but options include plain text and Ada data type statements.

Forms must be designed for fixed and variable length data fields.

- (see above)

The database of forms must be indexed by application and by type.

- This requirement is satisfied by the Forms Librarian.

The database of forms must be linked with data validation programs (when they exist).

- This requirement is satisfied by the Forms Librarian. Each form will include a (possibly null) entry naming the data validation program(s). Note that this is not a direct link in that automatic invocation of data validation programs is not provided.

The database of forms must be linked with appropriate run-time libraries for the terminal devices and the computer systems in use.

- Note that maintaining the forms data base is an independent function from terminal device support. There will be terminal support for a variety of terminal types. This will take the form of families of packages that are designed for specific terminal types. In most cases, the only dependent package is the Screen_Primitives package. However, the 3270 terminals may require more extensive terminal dependent code. In any case, terminal device controls are independent to the form definitions.

The form designer must be able to edit forms in the database and maintain version control.

- The Forms Editor interacts with the Forms Librarian. The Forms Librarian provides version control.

The screen/menu designer must be able to provide default values for all user responses.

- The form description includes default values for all components. The Forms Editor provides access to these fields.

The applications programmer must be able to directly control the sequencing of screens.

- Invoking a containing box results in a sequencing according to how subordinate boxes are located in the box hierarchy. However, the applications programmer may also invoke specific user response boxes directly, and thus maintain explicit control of the order of invocation. Note that 3270 terminals may require a different mechanism.

The applications programmer must be able to specify menus that are inclusive or exclusive.

- The menu box descriptions include a flag indicating selection type. The screen manager uses the flag to control the action taken when a menu item is selected. Specifically, the selection of an item in an exclusive box causes the deselection of any other item. The selection of an item in an inclusive box does not deselect other items.

Functional User

Must be able to edit within datafields being created, but must not be able to change the pre-defined portions of the form.

A functional user must be able to control traversal of the screen.

- The screen manager provides function keys for NEXT FIELD and PREVIOUS FIELD. (Note that these keys are defeated if the applications program controls box invocation directly.)

A functional user must be provided a menu that includes prompts and help information that is pertinent to his/her form(s) of interest.

- Each box in a form includes a (possibly null) reference to a help box. The help box is actually a pop-up text box that is invoked when the functional user presses the HELP function key. When this occurs, the help box associated with the current entry box (i.e., the box in which the cursor is located), is invoked. If the current box does not have a help box, then the help box of the current box's parent is searched. This continues until a help box is found or the top of the hierarchy is found.

A functional user must be able to recall, display and/or print any single form or all forms completed during a session.

- The Screen Manager includes several procedures for manipulating a data file, including a procedure for reading an arbitrary transaction.

A functional user must be warned with an audible sound or visible signal when he/she has exceeded the space allowed for a pre-defined data field and when the transaction space has been completed (when a form is filled) and a user action is required.

- The Screen Manager precludes the user from exceeding an entry field by either directly controlling the cursor (on ANSI terminals) or instructing the 3270 on the location of the modifiable and restricted fields.

Keystrokes will be minimized for invoking menus or forms, for data entry, and for editing.

- This is a general requirement that is satisfied by all interactive modules in the system.

Uniform menus must be displayed with illogical choices precluded.

- This requirement is satisfied with the form definition and the Screen Manager. Specifically, menu items may have one of several modes. The unavailable mode means that the item cannot be chosen and is skipped over when the end user traverses the menu with the cursor control keys. If an item is available, then it may be "current" if the cursor is positioned on that item. Also, if an item is available, it may be "selected." It is still the responsibility of the applications program (i.e., the program that calls the Screen Manager) to determine which items are available and notify the Screen Manager.

The cursor will be restricted to valid fields for data entry and menu selection

- The requirement is satisfied with the Screen Manager (see above).

Technical Environment

The Forms Management System must be designed to operate on VT-100 and IBM 3270 and IBM-PC work stations.

- This requirement is satisfied with the Screen Manager, Screen Controller, and Screen Primitives packages. Both the VT-100 and IBM-PC work stations respond to ANSI standard escape sequences. The inclusion of 3270 terminals brings up a number of (as yet) unanswered questions. Specifically, the 3270 is programmed with respect to prompt and entry fields. Data is not transferred to the host computer until the end user presses the ENTER key. This precludes the character by character validation possible on the other terminal type (e.g., for character field masking). It is not yet clear whether the 3270 can be programmed to work with menus in a manner consistent with the design of the Forms Management System. (At this stage, it seems desirable to provide as much functionality as possible. This may result in different terminals behaving differently, however, this is not unreasonable so long as it is understood that such differences are minor and are a result of limitations in terminal capabilities.)

Dependencies on operating system facilities will be isolated in modules that are part of the run-time library for the application program.

- Operating System dependent code is isolated. However, there is little operating system requirements. Opening and closing files are standard Ada procedure calls.

The Forms Management System must be capable of exploiting a color monitor.

- This requirement is satisfied by the Screen Manager. See the discussion on methods of accommodating various terminal types. (Note that the inclusion of this requirement means that monochrome terminals will behave differently than color terminals.)

Modifications to screen appearance should not require a change to the application program that uses that screen.

- The requirement is satisfied by the use of a form description. The Screen Manager reads display attributes from the Forms Library. Any changes to display attributes require no changes to an applications program. Furthermore, the Screen Manager provides functions for requesting information on the structure of a form. Thus, it is possible for an applications program to be designed such that it can adapt to any structure without modification.

Distribution List for IDA Memorandum Report M-366

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Ms. Sally Barnes DLA-ZWS HQ Defense Logistics Agency Cameron Station Alexandria, VA 22304-6100	10 copies
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies
IIT Research Institute 4550 Forbes Blvd., Suite 300 Lanham, MD 20706	1 copy
CSED Review Panel	
Dr. Dan Alpert, Director Center for Advanced Study University of Illinois 912 W. Illinois Street Urbana, Illinois 61801	*1 copy*
Dr. Barry W. Boehm TRW Defense Systems Group MS 2-2304 One Space Park Redondo Beach, CA 90278	1 copy
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1 copy
Dr. Larry E. Druffel Software Engineering Institute Shadyside Place 480 South Aiken Av. Pittsburgh, PA 15231	1 copy

NAME AND ADDRESS	NUMBER OF COPIES
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1 copy
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1 copy
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1 copy
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1 copy
IDA	
General W.Y. Smith, HQ	1 copy
Mr. Seymour Deitchman, HQ	1 copy
Mr. Philip Major, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Dr. David J. Carney, CSED	2 copies
Ms. Audrey A. Hook, CSED	2 copies
Ms. Katydean Price, CSED	2 copies
IDA Control & Distribution Vault	3 copies